

Android Security Framework: Enabling Generic and Extensible Access Control on Android

*Michael Backes, Sven Bugiel, Sebastian Gerling,
Philipp von Styp-Rekowsky*

Technischer Bericht Nr. A/01/2014

Android Security Framework: Enabling Generic and Extensible Access Control on Android

Michael Backes, Sven Bugiel, Sebastian Gerling, Philipp von Styp-Rekowsky
{backes,bugiel,sgerling,styp-rekowsky}@cs.uni-saarland.de
Saarland University/CISPA, Germany

Abstract

We introduce the *Android Security Framework (ASF)*, a generic, extensible security framework for Android that enables the development and integration of a wide spectrum of security models in form of code-based security modules. The design of ASF reflects lessons learned from the literature on established security frameworks (such as Linux Security Modules or the BSD MAC Framework) and intertwines them with the particular requirements and challenges from the design of Android’s software stack. ASF provides a novel security API that supports authors of Android security extensions in developing their modules. This overcomes the current unsatisfactory situation to provide security solutions as separate patches to the Android software stack or to embed them into Android’s mainline codebase. As a result, ASF provides different practical benefits such as a higher degree of acceptance, adaptation, and maintenance of security solutions than previously possible on Android. We present a prototypical implementation of ASF and demonstrate its effectiveness and efficiency by modularizing different security models from related work, such as context-aware access control, inlined reference monitoring, and type enforcement.

1 Introduction

For several decades now, the need for operating system security mechanisms to provide strong security and privacy guarantees has been well understood [28, 45, 31, 5]. Yet, recent attacks against smartphone end-user’s privacy and security [21, 56, 55, 38, 40, 9, 39] have shown that the fairly new smart device operating systems fail to provide these strong guarantees, for instance, with respect to access control or information flow control. To remedy this situation, security research has proposed

a wide spectrum of security models and extensions for mobile operating systems, most of them for the popular open-source Android OS. These extensions range from context-related access control [10], to developer-centric security policies [35] and dynamic, fine-grained permissions [34, 2, 25, 57], to domain isolation [7, 6], and type enforcement [47, 8].

However, the lack of a comprehensive security API for the development and modularization of security extensions on Android has created the unsatisfactory situation that all of these novel and warranted security models are either provided as model-specific patches to the Android software stack, or they became an integrated component of the Android OS design [47]. When considering the body of literature on established security frameworks, such as *Linux Security Modules (LSM)* [53] or the *BSD MAC Framework* [51], their history has taught that the need to patch the OS or the hardwiring of a specific security model impairs both the practical and theoretical aspects of security solutions. First, there is in general no consensus on the “right” security model, as demonstrated by the broad range of Android security extensions [10, 35, 2, 57, 7, 6, 47]. Thus, OS security mechanisms should not limit policy authors to one specific security model by embedding it into the OS design. Second, providing security solutions as “*security-model-specific Android forks*” impedes their maintainability across different OS versions, because every update to the Android software stack has to be re-evaluated for and applied to each fork separately.

Contributions. In this paper, we propose the design and implementation of an **ANDROID SECURITY FRAMEWORK (ASF)** that allows security experts to develop and deploy their security models in form of modules (or “security apps”). This provides the means to easily extend the Android security mecha-

nisms and avoids that policy authors have to choose “the right Android security fork” or that the OS vendor has to impose a specific security model. In the design of ASF we transfer the lessons learned and guiding principles from the literature on established OS security infrastructures to Android and intertwine them with new requirements for efficient security policies for multi-tiered software stacks of smart devices. This design solves a number of challenges in establishing a generic and extensible security framework for Android and we make the following concrete contributions:

1. Policy-agnostic, multi-tiered security infrastructure: The security infrastructure must avoid committing to one particular security model and enable authors of security extensions to develop as well as deploy their solutions in form of code. This requires special consideration of Android’s multi-tiered software stack and the dominant programming languages at each layer of this stack. For ASF we solve this by integrating security-model-agnostic enforcement hooks into the Android kernel, middle-ware and application layer and exposing these hooks through a novel security API to module authors.

2. Enabling edit automata policies: Various Android security solutions realize edit automata policies that not only truncate but also modify control flows. In ASF, the application layer and middle-ware hooks are specifically designed to allow module authors to leverage the rich semantics of Android’s application framework and to implement their security policies as edit automata. This required a re-thinking of the “classical” object manager design from the literature by shifting the edit automata logic from the infrastructure into the security modules.

3. Instantiation of existing security models: We demonstrate the efficiency and effectiveness of our ASF by instantiating different security models from related work on type enforcement [8, 47], context-related access control [10], Chinese Wall policies [6], and inlined access control [2] as modules.

4. Maintenance benefits for security extensions: Our ported security modules show how ASF simplifies maintainability of security extensions across different OS versions by shifting the bulk of effort to the security framework maintainer. This is similar to the maintenance of the application framework for regular apps. Hence, a comparable benefit to regular apps [22] in adaption and stability across OS versions can be expected of security modules.

5. Research and development benefits: We postulate that developing security solutions against a well documented security API also greatly contributes to *a)* a better understanding and analysis of

new security models that form a self-contained unit instead of being integrated to various components of the Android software stack, *b)* a better reproducibility and dissemination of new solutions since modules can be easily shared and instantiated, and *c)* a more convenient application of security knowledge to the Android software stack without the requirement to be familiar with the deep technical internals of Android.

Outline. The remainder of this paper is structured as follows. In Section 2 we provide necessary technical background information on Android’s design and security philosophy. We survey closest related work in Section 3 and derive from this design principles for a generic security framework on Android in Section 4. We present the design and implementation of ASF in Section 5 and show the instantiations of different security models from related work in Section 6. In Section 7 we evaluate our framework in terms of performance impact and discuss limitations of our approach. We conclude in Section 8.

2 Background on Android

In this section we provide necessary technical background information on Android.

2.1 Primer on Android

Android is an open-source software stack for embedded devices. The lowest level of this stack consists of a Linux kernel responsible for elementary services such as memory management, device drivers, and an Android-specific lightweight inter-process communication called Binder. On top of the kernel lies the extensive Android middleware, consisting of native libraries (e.g., SSL) and the application framework. System services in the middleware implement the bulk of Android’s application API (e.g., the location service) and pre-installed system apps at the application layer, like Contacts, complement this API.

Although application layer and middleware apps and services are commonly written as Java code, they are compiled to *dex* bytecode and run inside the *Dalvik Virtual Machine* (DVM). In addition to dex bytecode, apps and services can use native code libraries (i.e., C/C++) for low-level interactions with the underlying Linux system. Native code can be seamlessly integrated into dex bytecode by means of the *Java Native Interface*.

Android apps are generally composed of different components. The four basic app components are *Activities* (GUI for user interaction), *BroadcastReceivers* (mailbox for broadcast *Intent* messages),

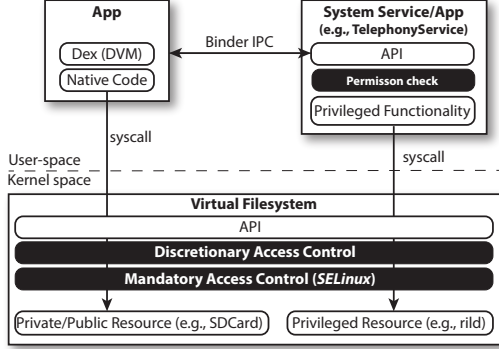


Figure 1: Android’s security architecture.

ContentProviders (SQL-like data management), and *Services* (long running operations without user interaction). All components can be interconnected remotely across application boundaries by using different abstractions of Android’s Binder IPC mechanism, such as *Intent* messages.

2.2 Android’s Security Philosophy

Android’s security philosophy dictates that all apps are sandboxed by executing them in separate processes with distinct user IDs (*UID*) and assigning them private data directories on the filesystem.

To achieve privilege separation between apps, Android introduces *Permissions*, i.e., privileges that an app is granted by the user at install-time. Without any permissions an app is not able to access security and privacy sensitive resources. Permissions are assigned to the app’s *UID* and enforced at two different points in the system architecture, as depicted in Figure 1: First, every application sandbox can directly interact with the underlying kernel through system calls, for instance, to edit files or open a network socket. These resources are either of private nature (i.e., are within the app’s private directory) or public resources (e.g., SDCard). Access control in the filesystem ensures that the apps’ processes have the necessary rights (i.e., Permissions) to issues particular syscalls, e.g., to open a file on the SDCard. The filesystem access control consists of the traditional Linux Discretionary Access Control, which is complemented (since Android v4.3) by SELinux based Mandatory Access Control (MAC).

Second, apps can interact through the Android API in a strictly controlled manner with highly privileged resources. To ensure system security and stability, apps are prohibited to access these highly privileged resources directly. Instead, those resources are wrapped by system services and apps that implement

the API. For instance, the *TelephonyService* communicates on behalf of apps with the radio interface layer daemon (*rild*) to initiate calls or send text messages. Whether an app is sufficiently privileged to successfully call the API is determined by a Permission check within the system services/apps. For this check, the Binder mechanism provides to the callee (system service/app) the *UID* of the caller (app).

3 Related Work

We first provide a synopsis of the development of extensible kernel security frameworks and discuss afterwards the current status of security extensions and frameworks for the Android software stack.

3.1 Extensible Kernel Access Control

The importance of the operating system in providing system security has been very well studied in the last decades [45, 28, 5, 31] and different approaches to extending operating systems with access control and security policies have been explored. These include system-call interposition [17, 41], software wrappers [18], and extensible access control frameworks like *Domain and Type Enforcement* (DTE) [4], *Generalized Framework for Access Control* (GFAC) [1], and *Flask* [49]. To realize these solutions, DTE has been provided as a patch to the UNIX system [3], while GFAC and Flask have been implemented as patches to the Linux kernel by the RSBAC [36] and SELinux [30] projects. However, this led to an intricate situation: On the one hand, maintaining these solutions as patches incurred high maintenance costs for adapting the patches to kernel changes. On the other hand, none of these solutions was included in the vanilla kernel because this would constrain security policy authors to one specific security model. This constrain would be unsatisfying since there exists in general no consensus on the “right” security model. To remedy this situation, extensible security frameworks have been proposed [53, 51] that allow the extension of the system with trusted code modules that implement specific security models. Module authors are supported with an API that exposes kernel abstractions as well as operations and facilitates the implementation of the desired security architecture and model. The results of this research have been integrated into the mainline kernels as the *Linux Security Modules* framework (LSM) [53] and the *BSD MAC Framework* [51]. Additionally, different access control models, such as SELinux type enforcement [48] or TOMOYO path-based access control [23], have been ported as modules on top of the

LSM and the BSD MAC framework.

3.2 Android Security

Security research and reported attacks have shown that the stock Android security mechanisms are insufficient to protect the end-user’s privacy and the system security. At the same time, research has also proposed different security extensions for Android to mitigate these attacks and to improve protection of the end-user’s privacy. The spectrum of the proposed solutions covers a wide range of security models and architectures. To name a few: *CRePE* [10] provides a context-related access control, where the context can be, e.g., the device’s location. *Saint* [35] enables developer-centric policies that allow app developers to ship their apps with rules that regulate the app’s interactions with other apps and can thus protect the app from misuse and attacks. Different approaches to more dynamic and fine-grained permissions have been proposed based on system-centric enforcement (*Apex* [34] and *TISSA* [57]) or inlined reference monitors (*Dr. Android and Mr. Hide* [25], *Aurasium* [54], and *AppGuard* [2]). *XManDroid* [6] enforces Chinese Wall policies to prevent confused deputy and collusion attacks. *TrustDroid* [7] and *MOSES* [44] isolate different domains such as “Work” and “Private” from each other. *SE Android* [47] and *FlaskDroid* [8] bring type enforcement to Android, where SE Android focuses on the kernel layer and has been partially included into the mainline Android source code, and FlaskDroid extends type enforcement to Android’s middleware layer on top of SE Android.

4 Motivation and Requirements Analysis

The current development of Android security extensions has strong parallels to the initial development of the above mentioned Linux and BSD security extensions, since current Android security extensions are provided as patches to the software stack or, in the case of SE Android [47], are embedded into the Android source tree. For the same, above mentioned reasons as for the early Linux and BSD security extensions, this impedes the applicability and adaption of Android security extensions and additionally precludes many of the benefits that a modular composition could offer in terms of maintenance: Embedding SE Android’s security model into Android’s source tree limits policy authors to the expressiveness and boundaries of type enforcement, whereas provisioning security models and architectures as patches to Android’s software stack forces policy authors to chose

a solution-specific Android fork. This requires for every version update to the Android OS a re-evaluation and port of each separate fork. Moreover, security solutions cannot be easily compared with each other, because their infrastructures are deeply embedded into the Android software stack.

In this paper, we develop in the spirit of the two de facto most established security frameworks, *Linux Security Modules* (LSM) [30] and the *BSD MAC Framework* [51, 52], a generic and extensible ANDROID SECURITY FRAMEWORK that allows the instantiation and deployment of different security models as loadable modules at Android’s application layer, middleware, and kernel. The two most important guiding principles from LSM and the BSD MAC framework that govern the design of our ANDROID SECURITY FRAMEWORK are: 1) provisioning of policies as code instead of data; and 2) providing a policy-agnostic OS security infrastructure. In the remainder of this section, we analyze the requirements and challenges for their transfer to the Android software stack.

Policy as code and not data. The first guiding principle is that policies should be supported as code instead of data (such as rules written in one predetermined policy language). Providing an extensible security framework that supports loading of policy logic as code avoids committing to one particular security model or architecture. For Android, this removes the need to chose a particular extension-specific Android fork or to be limited to one specific security model in the mainline Android software stack. Additionally, developing modules against an OS security API provides the benefits of modularization for developing and maintaining security extensions. This includes, foremost, a higher functional cohesion of security modules and lower coupling with the Android software stack and, hence, can significantly reduce the maintenance overhead of modules, especially in case of OS changes. Moreover, it allows a better dissemination, comparison, and analysis of self-contained security modules.

Transferring this principle to an extensible security framework for Android poses the additional requirement to consider the semantics and dominant programming languages of the different layers of Android’s software stack. LSM and the BSD MAC Framework, for instance, as part of the Linux and BSD kernels, support modules written in C and operate on kernel data structures (e.g., filesystem inodes). While this applies to the Android Linux kernel as well, an Android security framework should additionally support modules written for Android’s semantically-rich middleware and application layers. That means

modules written in Java and operating on application framework classes (e.g., Intents or app components).

Policy-agnostic security infrastructure. The second principle is that the security framework and its API should be policy-agnostic. This means that the different layers of the software stack are aware of the security infrastructure, but policy-specific intrusions into these layers are avoided and policy-specific data structures and logic are confined to security modules.

A particular additional requirement for a security framework on Android are enforcement hooks in the middleware and application layer that support *edit automata* [27] policies, as promoted by different solutions [57, 24, 7, 25, 2]. Edit automata, in contrast to truncation automata, can not only abort control flows but also divert or manipulate them and, thus, give policy authors a higher degree of freedom in implementing their enforcement strategies. For instance, when querying a ContentProvider component, the policy could simply deny access by throwing a Java SecurityException (truncation), but also modify the return value to return filtered, empty, or fake data (edit). To technically enable security modules to implement edit automata, our design requires a re-thinking of the “classical” object manager vs. policy server design that is used, e.g., in LSM. Object managers (i.e., enforcement points) are responsible for assigning security labels to the objects that they manage and for both requesting and enforcing access control decisions from the policy server (i.e., policy decision point). Because this design embeds the enforcement logic into the system independently from the security model, it is unfit for realizing edit automata. Thus, our design requires hooks that generically support different enforcement strategies and shift the enforcement and object labelling logic from the object managers to the security modules.

Moreover, the security framework should provide a policy-agnostic infrastructure for common operations such as event notifications or module life-cycle management—thus reducing the overhead for policy authors to implement common functionality.

5 Android Security Framework

In the following we present ASF. We provide further technical details in the appendices of this submission.

5.1 Framework Overview

The basic idea behind our ANDROID SECURITY FRAMEWORK is to extend Android with a new security API that incorporates the design principles

explained in Section 4. This API allows to easily author, integrate, and enforce generic security policies. Figure 2 provides an overview of our ASF and we explain its building blocks in the following.

5.1.1 Reference Monitors

In our design we differentiate between policy enforcing code and policy decision making code. For enforcement we use reference monitors [26, 11] at all layers of the Android software stack, i.e., at the application layer, the middleware layer, and the kernel layer. Each reference monitor protects one specific privileged resource and is placed such, that it is always invoked by the control flow between the Android API and access to the resource. The benefit of this multi-tiered enforcement is that each reference monitor can operate with the semantics of its respective layer. In conjunction all monitors enable exceedingly powerful and semantically rich security policies.

5.1.2 Security Modules

Security extensions are deployed in the form of code modules and loaded into the security frameworks at the middleware and kernel level. Each module implements a policy engine that manages its own security policies and acts as policy decision making point. Security modules are integrated into the security frameworks through a security API that exposes objects and operations of the different software stack layers. As part of this API, each module implements an interface for enforcement functions (i.e., functions that make a policy decision for a particular reference monitor in the system), management functions (e.g., module life-cycle events), and other interfaces that will be explained in Section 5.2.

To provide a clear separation between policy decision logic using kernel level semantics and logic using middleware/application layer semantics, each module consists of two sub-modules: a KERNEL SUB-MODULE leveraging the already existing Linux Security Module (LSM) infrastructure of the Linux kernel and a MIDDLEWARE SUB-MODULE, for which we designed and implemented a novel security infrastructure at the application and middleware layers.

5.1.3 Front-end Apps

To enable user configurable policies or graphical event notifications, modules might want to include user interfaces. To this end, the module developers (or external parties being aware of the modules) can deploy standard Android apps that act as front-end apps and communicate through the framework API

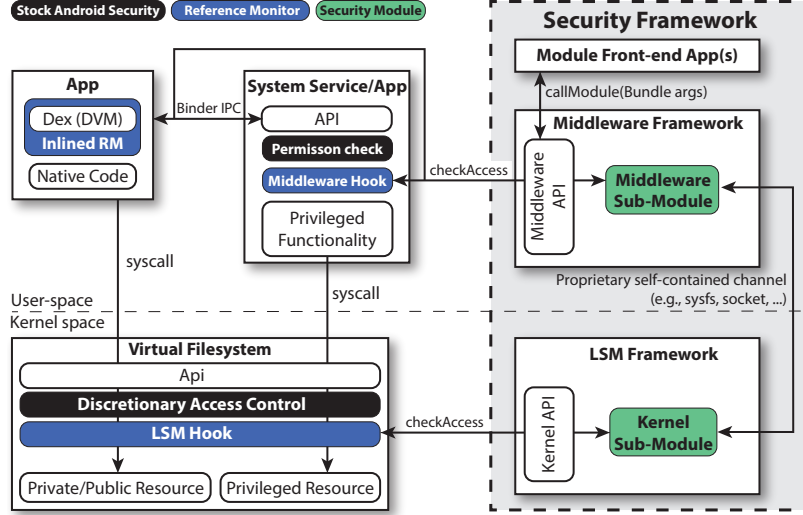


Figure 2: Android Security Framework architecture.

directly with the module. In our framework API, we enable this communication through a *Bundle* based communication protocol. A *Bundle* is a key-value store that supports heterogeneous value types (e.g., Integer and String) and that can be transmitted via Binder IPC between the app process and the module. It is the responsibility of the module to verify that the caller has the necessary privileges to issue commands.

5.2 Framework Infrastructure

We present now in a bottom-up approach details about the ASF infrastructure that has been prototypically implemented for Android v4.3 and currently comprises 4606 lines of code.

5.2.1 Kernel Space

At the kernel level we employ the existing Linux Security Module (LSM) [53] framework of the Linux kernel. LSM implements an infrastructure for mandatory access control at kernel level and provides a number of enforcement hooks within kernel components such as the process management, the network stack, or the virtual filesystem. The **KERNEL SUB-MODULE** is implemented as a standard Linux Security Module that registers through the LSM API for the LSM hooks in the system. This enables the implementation of a variety of different security models that operate with kernel level semantics. Modules include, for instance, SELinux type enforcement [48], which uses security labels attached to filesystem inodes and process data structures, or TOMOYO [23] for path-based access control, or custom modules [33].

Kernel-level policies form truncation automata that terminate illegal control flows, e.g., on access to files.

Since there might be operational interdependencies between the **KERNEL SUB-MODULE** and user-space processes (e.g., re-labelling the security context of files or propagating kernel access control decisions to the **MIDDLEWARE SUB-MODULE**), the kernel module can implement proprietary channels for communication between kernel- and user-space (e.g., sockets or sysfs entries).

5.2.2 Middleware Layer

At the middleware layer we extended the system services and apps that implement the Android API with hooks that enforce access control decisions made by the **MIDDLEWARE SUB-MODULE**. The middleware security framework is executed as a new Android system service and mediates between our hooks and the **MIDDLEWARE SUB-MODULE**. In contrast to previous solutions for (generic) access control on Android, our hooks are policy-agnostic and not tailored to one specific security model. Each hook takes as arguments all relevant, ambient information of the current control flow that led to the hook’s invocation. For instance, Listing 1 presents two exemplary hooks in our system: one for the Intent broadcasting subsystem of the *ActivityManagerService* (line 1) and one for the *LocationManagerService* that implements the location API of Android (line 2). Both provide to the **MIDDLEWARE SUB-MODULE** information about the current caller to the Android API, i.e., **APP** in Figure 2 (parameters `callingUid` and `callingPid`). However, all other parameters are specific to the

Listing 1: *Exemplary enforcement functions*

```
1 public boolean
   security_broadcast_deliverToRegisteredReceiver
   (Intent intent, ComponentName targetComp,
    String requiredPermission, int targetUid, int
    targetPid, String callerPackage, ApplicationInfo
    callerApp, int callingUid, int callingPid);
2 public Location security_location_getLastLocation
   (Location currentLocation, LocationRequest
    request, int callingUid, int callingPid);
```

hooks’ contexts, e.g., the hook in line 1 provides information about the Intent being broadcasted (parameter `intent`) and the app component that should receive this Intent (parameters `targetComp` through `targetPid`). Thus, the hooks support policies that use the rich middleware-specific semantics.

In general, all hooks support truncation automata as policies by either allowing the module to throw exceptions that terminate the control flow and that are returned to the caller of the Android API, or by explicitly requiring a boolean return value that indicates whether the hook truncates the control flow or not (line 1 in Listing 1). A subset of the hooks additionally supports edit automata policies, that is the module can modify or replace return values of the Android API function or modify/replace arguments that divert or affect the further control flow after the hook. For instance, the *LocationManagerService* hook in Listing 1 (line 2) allows the module to edit or replace the Location object that is returned to the app that requested the current device location.

We describe the API of the middleware framework as well as the detailed structure of modules in the next sections. Appendix B provides an overview of the coverage of our current enforcement hooks.

5.2.3 Application Layer

At the application layer, our ANDROID SECURITY FRAMEWORK provides a mechanism to inject access control hooks into apps themselves. This access control technique is based on the concept of *inlined reference monitors* (IRM) pioneered by Erlingsson and Schneider [16]. The basic idea is to rewrite an untrusted app such that the reference monitor is directly embedded into the app itself, yielding a “self-monitoring” app. The main advantage of policy enforcement in the caller’s process context is that the hook and transitively the security module has full access to the internal state of the app and can thus provide rich contextual information about the caller. For instance, by inspecting the call stack, the IRM is able to distinguish between calls from different app components, e.g., advertisement libraries, thereby

allowing more expressive and fine-grained access control policies. ASF provides an instrumentation API that enables security modules to dynamically hook any Java function within an app’s DVM. Hooked functions divert the control flow of the program to the reference monitor, which thereby not only gains access to all function arguments but can also modify or replace the function’s return value. Thus, the IRM is also able to enforce edit automata security policies. Furthermore, in contrast to the hooks placed in the Android middleware, application layer hooks are dynamic: Hooks are injected by directly modifying the target app’s DVM memory when a new app process is started. This design enables security modules to dynamically create and remove hooks at runtime as well as to inject app-specific hooks.

5.3 Middleware Framework API

We elaborate now in more detail on our framework API and the interaction between modules and the security infrastructure. Since we use the LSM framework as is, we focus here on our newly introduced middleware security framework and refer to the kernel documentation [29] for details on the LSM API.

The middleware framework API of our current implementation contains 168 functions. A full listing of our current API is provided in Appendix A. This API can be broken down into the following categories:

Enforcement functions. These module functions form the bulk of the API (136 methods) and are called by the framework whenever the enforcement hooks in system apps and services are triggered. Each hook has a corresponding function in the module API that implements the policy decision logic for this hook. Enforcement functions have the same method signatures as their hooks (cf. Listing 1), i.e., all parameters provided by a hook are passed to its enforcement function. Passing arguments by reference or expecting objects as return values allows these functions to implement edit automata logic.

Kernel Sub-Module Interface. To avoid policy-specific interfaces for the communication between middleware/application layer apps and the KERNEL SUB-MODULE, we introduce a generic kernel module API as part of the middleware framework API. It allows apps and services a controlled access to Linux security modules (cf. Listing 3 in Appendix A). Each security module can implement this interface and internally translate the API calls to calls on the proprietary channel between the user-space and the Linux security module. Two particular challenges

for establishing this interface were the self-contained security checks of the kernel module and the requirement that this interface is already available during system boot. To guarantee security, the kernel module is required to perform policy checks to verify that a user-space process is sufficiently privileged to issue commands to it. Additionally, the kernel module is called before the middleware framework can load any MIDDLEWARE SUB-MODULE, e.g., it can be called by Zygote when spawning new app processes. To solve these challenges, our design avoids an additional layer of indirection (i.e., IPC) for communication with the kernel module and loads the interface implementations via the Java reflection API statically into the application framework when it is bootstrapped. This ensures that the calling processes communicate directly with the kernel module through our generic API and that the kernel module can be called independently of middleware services.

Life-cycle management. Every module must implement functions for life-cycle management, such as initialization or shutdown. This enables the framework to inform the module when the system has reached a state during the boot cycle from which on the module will be called or when the system shuts down. Modules should use these functions, e.g., to initiate their policy engines or to save internal states to persistent storage before the device turns off.

Event notifications. Event notification interfaces are used to propagate important system events to the module. For instance, modules should be immediately informed when an app was successfully installed, replaced, or removed. Although this information is usually propagated via a broadcast Intents, the time gap between package change and broadcast delivery might cause inconsistencies in module states. Hence these events must be delivered synchronously.

Framework Callbacks. The framework provides modules a callback interface for communicating in a more direct manner with system services, such as the *PackageManagerService*, and avoids the need to go through the Android API. This is desirable for policy authors that want to leverage the middleware internal information for a more efficient access control enforcement. Our current callback interface, for instance, includes functions that allow modules to efficiently resolve PIDs to application package names.

Proprietary protocols. We introduced in our framework API a *callModule()* function that allows

modules to implement proprietary communication protocols with other apps that are aware of this specific module, e.g., the front-end apps (cf. Section 5.1). When using *callModule()*, these protocols are based on *Bundles* and enable a protocol similar to the Parcel-based Binder IPC: apps serialize function arguments to a Bundle and add an identifier for the function the receiver should execute with the deserialized arguments.

IRM Instrumentation. The framework provides an instrumentation API that enables security modules to hook any Java function within selected app processes. Function calls can be redirected to an inlined reference monitor that enforces policy decisions made by the module. Appendix F explains an example that uses the instrumentation API.

Hooks injected via the instrumentation API are local to the app process that the API is called from. Therefore, all calls to the instrumentation API need to be performed from within a target application’s process. We solve this by placing an instrumentation hook in the *ActivityManagerService* that is triggered when a new app process is about to be launched. A module that implements this hook has to return the name of a module’s Java class (the inlined reference monitor) that will be executed within the app’s process before control flow is passed to the app itself. This is accomplished by modifying the arguments passed to Zygote: Instrumented apps are started via a special wrapper class that loads and executes the instrumentation code before running the app.

5.4 Middleware Security Modules

We elaborate in more detail on the structure of security modules. Again, we use Linux security modules as is [29] and, thus, focus here on the MIDDLEWARE SUB-MODULE. A middleware security module is, simply speaking, an app that is created with an Android SDK that includes our new security API. It is deployed to a protected location on the file system, from where it is loaded during boot. The module package is a *Jar* file that contains all the module’s program code, resources, and manifest file (cf. Figure 3):

Module Manifest. The manifest (formatted in XML) declares properties such as the module author or code version, and, more importantly, the name of the main Java class that forms the entry point for the module.

Classes.dex. The *classes.dex* file contains, as in regular Android apps, the Java code compiled to

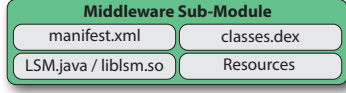


Figure 3: *Middleware security module structure.*

Dalvik executable bytecode (DEX). It contains all Java classes that implement the security module’s logic. During the load process of the MIDDLEWARE SUB-MODULE, the middleware framework uses the Java reflection API to load the module’s main class (as specified in the manifest) from *classes.dex*. To ensure that the reflection works error-free, the main class must implement the API as described in Section 5.3 (and listed in Appendix A). Since the API defines currently more than a hundred methods, but a security module very likely requires only a subset of those, our SDK provides an abstract class that implements the API. That abstract class can be sub-classed by the module’s main class, which then only needs to override the required functions. The abstract class returns for each non-overridden enforcement function an allow decision.

LSM interface. The proprietary interface between the user-space processes and the Linux security module in the kernel is implemented through a native library *liblsm.so* and a corresponding Java class *LSM.java*, which exposes the native library via the Java Native Interface. *LSM.java* has to implement the generic interface for the communication with the kernel that was explained in the previous section. The generic kernel module interface of ASF (called *KMAC.java*) loads *LSM.java* through the Java reflection API into Android’s application framework. This allows apps and services to communicate via *KMAC* (and reflectively through *LSM.java*) with the kernel module and avoids a policy-specific interface. We exemplify this mechanism in Appendix E by integrating SELinux into Zygote.

Resources. Each module can ship with proprietary resources, such as initial configuration files or required binaries (e.g., *csstools* for TOMOYO [23]). During module instantiation, the framework informs the module about the filesystem location of its Jar file, enabling the module to extract these resources on-demand from its file.

5.5 Support for Stackable and Dynamical Loadable Modules

Finally, two desirable properties for implementing an extensible security framework such as our ASF are dynamically loadable policies and policy composition (i.e., stacking modules). In the following we explain why we chose to *permit* these features by design, but *not* consider them a requirement for our solution.

Dynamically Loadable Modules. Being able to dynamically load and unload modules is desirable, for instance, to speed up the development and testing cycles of modules and, in fact, we used this feature during the development of our example use-cases (cf. Section 6). However, the arguments to support dynamically loadable modules beyond development are disputed: First, dynamic loading is not always technically possible. A small set of static policy models, such as type enforcement [47, 8], require that all subjects and objects are labeled with a security context. Supporting such extensive labeling operations at runtime is an intricate problem. Second, there exist security considerations. The loading and unloading of modules must be strictly controlled to ensure that only integrity protected, trusted modules are loaded. Otherwise, given the privileges of modules, this would open the way to powerful malware modules. In our design we agree with the conclusions of the various Linux security module authors [12] and consider the drawbacks of dynamically loadable modules to outweigh their benefits. Therefore, we load the module once during the system boot and *permit* users of our framework to additionally activate dynamic unloading and loading of modules. But we currently do not consider this feature a requirement for our solution. However, community and market forces (e.g., vendors of security solutions) have to determine whether there is a need for supporting dynamically loadable modules in the future (e.g., establishing a “security app market”) and, hence, whether we have to revise our requirements analysis.

Stackable Modules. Composing the overall policy from multiple, simultaneously loaded and independent policies is a desirable feature, since usually no “one-size-fits-all” policy exists. Android, for instance, implements currently a quadruple-policy approach consisting of Permissions, SE Android type enforcement, AppOps, and Linux capabilities—each being responsible for a different aspect of the overall access control strategy. Multiple policies will naturally conflict and thus require the security framework to support different policy composition and reconciliation

strategies (e.g., consensus or priority based) [43, 32]. However, supporting fully generic policy composition is quite a challenge and has been shown to be intractable [20]. Thus, despite its benefits, we decided in our design to follow the lessons learned by the LSM developers [53] and to only *permit* module developers to implement stackable modules, but we do not provide explicit interfaces for stacked modules in our framework infrastructure. The approach to stacking modules would be to provide a “composition module” that implements policy reconciliation and composition logic and which in turn can load other modules and multiplex API calls between them.

6 Example Security Modules

In this section, we demonstrate the efficiency and effectiveness of our ANDROID SECURITY FRAMEWORK by instantiating different security models from related work. To illustrate the versatility of ASF, we chose models from the areas of inlined reference monitoring, context-based access control, domain isolation, and type enforcement. We present further instantiations of other security models in Appendix H.

6.1 Inlined Reference Monitoring [2]

We use AppGuard [2] as the use-case to illustrate the applicability of our IRM instrumentation API, but similar application rewriting approaches [25] are also feasible. AppGuard is a privacy app for Android that enables end-users to enforce fine-grained access control policies on 3rd party apps by restricting their ability to access critical system resources. It does so by injecting an IRM into the apps themselves. This approach supports security policies not easily enforceable by traditional external reference monitors in the Android middleware or kernel, e.g., to enforce the use of *https* over *http*.

Implementation as a module: We ported AppGuard¹ as a module for ASF by separating its privacy app into three components: We adapted the (1) AppGuard reference monitor with its dynamic hook placement and policy enforcement logic to use the IRM instrumentation API provided by ASF. The reference monitor is injected into selected app processes via our framework at app startup. The policy decision logic and persistent storage of policy settings was moved into (2) a middleware module. The middleware module selects the apps into which the IRM is injected. It also implements a Bundle-based

communication protocol to exchange policy decisions and security events with the IRM component and with (3) a front-end app. The front-end app allows the user to adjust policy settings and to view logs of security-relevant events. We used the policies included in the original AppGuard implementation to confirm that policy enforcement by our security module and by the original implementation are identical.

Our AppGuard security module consists of 5059 LoC in total (cf. Table 1), with 782 LoC residing in the middleware module and 4277 LoC in the IRM. Our module diverts in 18.18% of all LoC from the original code. The majority of the difference, 728 LoC, is attributed to moving the policy decision logic into the middleware module, while only 46 LoC were required to adapt the inlined reference monitor to use the provided instrumentation API.

6.2 CRePE [10]

CRePE is a security extension to Android v2.3 that enforces fine-grained and context-related access control policies. The context is based on the geolocation of the device and, depending on this location, CRePE either allows or denies apps access to security and privacy sensitive information. The security policies can be deployed over different channels, e.g., via SMS. To enforce the policies, CRePE hooked all relevant system services, e.g., to override Android’s default permission check with its context-related check.

Implementation as a module: We ported CRePE² as a security module for ASF by moving its policy engine class *CRePEPolicyManagerService* and related classes, which were originally running as a separate system service, into a module (cf. Figure 4). On initialization, CRePE’s context detector registers as a listener for location updates to detect context changes. Additionally, we used the enforcement functions of our API to re-implement the logic of CRePE’s hooks. Furthermore, CRePE uses front-end apps to parse and inject policies from different channels. We moved the policy parser into the module and established a Bundle-based communication protocol between the front-end apps and the module to forward received policies for processing. We used the example policies shipped with the CRePE source code to successfully confirm that the enforcement by our module yields the same results as the original CRePE implementation.

Our port of CRePE as a security module consists of 3682 lines of code (cf. Table 1), excluding the un-

¹Source code provided by the original authors

²Source code retrieved from <http://sourceforge.net/projects/crepedroid>

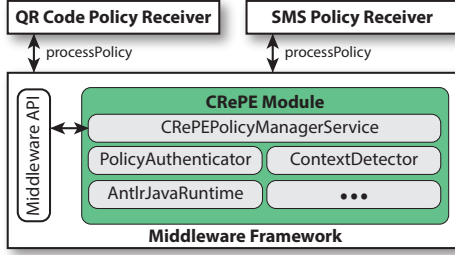


Figure 4: CRePE module

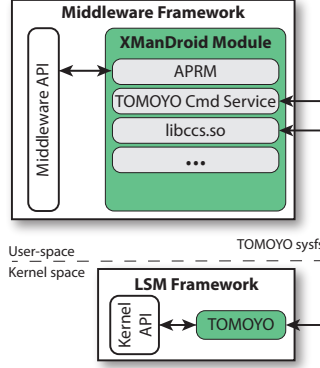


Figure 5: XManDroid module

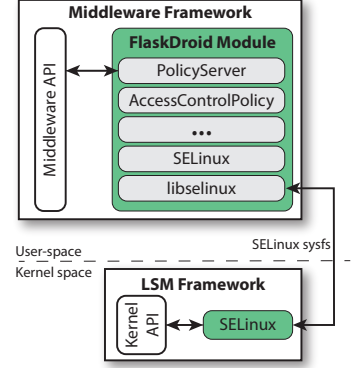


Figure 6: FlaskDroid module

Existing solution	LoC of module policy engine	LoC added/removed/edited (total delta)
AppGuard [2]	5059	+828/-79/o 13 (18.18%)
CRePE [10]	3682	+915/-48/o 45 (27.38%)
XManDroid [6]	3244	+153/-14/o 28 (6.01%)
FlaskDroid [8]	4968	+749/-32/o 40 (16.53%)

Table 1: Effort of porting different security extensions as module on our ANDROID SECURITY FRAMEWORK.

modified ANTLR runtime (7526 LoC). Of these 3682 LoC 27.38% were changed during the port. The bulk of this difference, 817 LoC, is attributed to relocating the policy parser. Implementing the Bundle-based communication protocol added 74 LoC. Only 2 LoC had to be changed to adapt CRePE’s calls to the Android API from its original Android v2.3 implementation to our Android v4.3 implementation.

6.3 XManDroid [6]

XManDroid extends the security architecture of Android v2.2.1 to enforce Chinese Wall policies between apps that might jointly leak privacy sensitive information. It uses hooks within different system services in the middleware and TOMOYO Linux at the kernel level to monitor all access control requests, reflect these interactions between processes/apps in a graph model, and use this model to check against policies whether an inter-app communication would lead to an attack state. If so, it denies the new communication. The policy decision logic is implemented as an extension (*APRM*) to the *ActivityManagerService*.

Implementation as a module: We ported XManDroid³ to a module for ASF (cf. Figure 5) by extracting the policy decision logic from the *ActivityManagerService* and moving it into a module. Using the enforcement functions of our API we moved the XManDroid hook logic to this module as well and, by

using a proprietary channel, we enabled the *APRM* to communicate from the module with the TOMOYO kernel module. The kernel was specifically compiled and deployed with a TOMOYO Linux security module. The XManDroid source code comes with an example configuration for the policy described in [6] and we used this configuration to successfully confirm that our module yields the same enforcement results as the original XManDroid implementation.

Our XManDroid middleware module consists of 3244 lines of code, excluding the unchanged JGraphT library (9256 LoC). Our module differs in only 6.01% (195 LoC) from the original implementation. Of these 195 LoC, 141 are attributed to additions necessary for porting XManDroid’s filtering logic for broadcasts from the *ActivityManagerService* to the module.

6.4 Type Enforcement [47, 8]

SE Android [47] brought SELinux type enforcement to the Android kernel and established the required user space support, e.g., it extended Zygote to label new app processes with a security type. FlaskDroid [8], developed for Android v4.0.3, extends SE Android’s type enforcement to Android’s middleware. Building on SEAndroid’s kernel and low-level patches, it adds policy-specific hooks as policy enforcement points to various system services and apps in Android’s middleware. The policy decisions at kernel level are made by the SELinux kernel module, while the decisions at middleware are made centrally in a policy server service. Both policy decision points

³Source code provided by the original authors

	Frequency	Mean (μ s)
Stock Android 4.3	7320	116.182 \pm 4.550
ASF v4.3	6009	129.851 \pm 5.681

Table 2: *Weighted average performance overhead of executing hooked functions in stock Android and in our ANDROID SECURITY FRAMEWORK. The margin of error is given for the 95% confidence interval.*

decide based on subject type, object type, and object class reported by the hooks at their respective layer whether control flows should be truncated or not.

Implementation as module: We realized type enforcement with our ASF by porting FlaskDroid⁴ as a module (cf. Figure 6). At kernel level, we use the SE Android kernel and provide an SELinux-specific interface implementation for the kernel module. A technical description of this interface implementation is provided in Appendix E. Further, we moved the middleware policy server and its dependencies into the middleware module. Using the enforcement functions of our API, we moved the policy-specific hook logic of FlaskDroid into the module as well (cf. Appendix G). Additionally, we used SE Android’s build system to label the file-system with security types.

Our port of FlaskDroid’s middleware component as a security module consists of 4968 lines of code (cf. Table 1) and differs in only 16.53% of all LoC from the original code. The bulk of these changes (550 LoC) is attributed to additions for implementing a mapping from the enforcement functions of our framework API to FlaskDroid’s type checks. To confirm the correct enforcement of policies, we used the policies for middleware and kernel level that are provided with the FlaskDroid source code. Additionally, we noticed during our tests that the original implementation contains an error in assigning middleware security types to processes.⁵ Additional changes were necessary to fix this error in our FlaskDroid module.

7 Evaluation and Discussion

In this section we evaluate the performance of our ANDROID SECURITY FRAMEWORK and discuss its current scope and prospective future work.

⁴Source code retrieved from <http://www.flaskdroid.org/>

⁵It maps process UIDs always to the security type of the first package in a shared sandbox, although the policy can define different types for packages that share a UID.

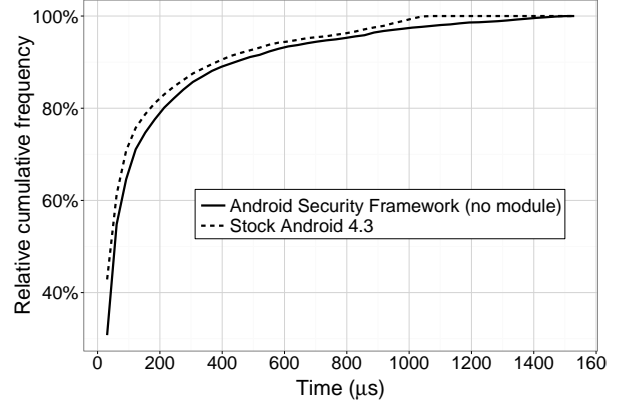


Figure 7: *Relative cumulative frequency distribution of micro benchmarks in stock Android (dashed line) vs. ANDROID SECURITY FRAMEWORK (solid line).*

7.1 Performance

Although the actual performance overhead strongly depends on the overhead imposed by the loaded module, we wanted to establish a baseline for the impact of our ANDROID SECURITY FRAMEWORK on the system performance. The performance of LSM has been evaluated separately, e.g., for SEAndroid [47], and we are interested here in the effect of our new middleware security framework on the performance of instrumented middleware system services and apps.

Methodology. We implemented our ASF as a modification to the Android OS code base in version 4.3_r3.1 (“Jelly Bean”) and used the Android Linux kernel in branch *android-omap-tuna-3.0-jb-mr1.1*. We performed micro-benchmarks for all execution paths on which a hook diverts the control flow to our middleware framework: We first measured the execution time of each hooked function with no security module loaded and allowing by default all access. Afterwards we repeated this test with hooks disabled to measure the default performance of the same functions and thus operating like a stock Android. All our micro-benchmarks were performed on a standard Nexus 7 development tablet (Quad-core 1.51 GHz CPU and 2GB DDR3L RAM), which we booted and then used according to a testplan for different daily tasks such as browsing the Internet, sending text messages and e-mails, contacts management, or (un-)installing 3rd party apps.

Micro-benchmark results. Table 2 presents the number of measurements for each test case and their mean values. To exclude extreme outliers, we excluded in both measurement series the highest decile

of the measurements. For ASF the mean is the weighted mean value with consideration of the frequency of each single hook. Table 4 in Appendix C provides a break down of the most frequently called hooked Android API functions and their mean execution time. In overall, our framework with no loaded module imposed with 129.851 μ s approximately only 11.8% overhead compared to stock Android. Figure 7 presents the relative cumulative frequency distribution of our measurements series and further illustrates this low performance overhead. Appendix D provides an overview of the micro-benchmark results for our example modules from Section 6.

7.2 Current Scope and Future Work

System setup. Certain security models require a preparatory system setup. For instance, type enforcement requires a pre-labelling of all subjects and objects as well as enabling the SELinux kernel module. After the system has been setup, ASF supports modularization of these security models (cf. Section 6.4).

Module Integrity. As part of the kernel, the KERNEL SUB-MODULE has the highest level of integrity. In contrast, the MIDDLEWARE SUB-MODULE, as a user space process, can be circumvented or compromised by attacks against the underlying system (e.g., root exploits) and thus requires support by the kernel modules to prevent low-level privilege escalation attacks. Inlined reference monitors are inherently susceptible to attacks by malicious applications, because the reference monitor executes in the same process as the application that it monitors and no strong security boundary exists between the monitor and the app code. To remedy this situation, we are currently retrofitting Android’s application model to combine the benefits of inlined and of system-centric reference monitors. By splitting apps into smaller units of trust (e.g., app components and ad libs), system-centric reference monitors are able to differentiate distinct trust levels within apps [42, 50, 46, 37].

Completeness. It is crucial for the effectiveness of our security framework, that *all* access to security and privacy sensitive resources is mediated by the reference monitors. We consider it out of scope for this submission to formally verify the completeness of our prototype framework, but plan to use recent advances in static and dynamic analysis on Android to verify the placement of our hooks, similarly to how it was done for the LSM framework [13, 19].

Information flow control. Our framework provides modules with the control over which subject (e.g., app) has *access to* which objects (e.g., device location), but it cannot control how privileged subjects *distribute* this information. Controlling information flows is an orthogonal problem specifically addressed by solutions such as *TaintDroid* [14], *AppFence* [24], or *MOSES* [44]. We plan to integrate such data flow solutions into our framework and to extend our security API with new generic calls for taint labeling and taint checking.

Framework Maintenance. ASF removes the burden for authors of security extensions to patch every Android OS version. Instead it puts this burden onto the framework maintainer, which is preferably the OS vendor like Google. In light of the vendor’s expert knowledge about the Android software stack, we consider the maintenance overhead for integrating ASF into the continuous integration of the Android development as moderate. In particular the recent development history of Android v4.x, which had one focus clearly set on security (e.g., integration of SELinux, use of capabilities, AppOps and IntentFirewall), makes us optimistic that the required commitment by the vendor is viable.

8 Conclusion

In this paper we presented the ANDROID SECURITY FRAMEWORK (ASF), an extensible and policy-agnostic security infrastructure for Android. ASF allows security experts to develop Android security extensions against a novel Android security API and to deploy their solutions in form of modules or “security apps”. Modularizing security extensions overcomes the current unsatisfactory situation that policy authors are either limited to one predetermined security model that is embedded in the Android software stack or that they are forced to confide in a security-model-specific Android fork instead of the mainline Android code base. Additionally, this modularization provides a number of benefits such as easier maintenance and direct comparison of security extensions. We demonstrated the effectiveness and efficiency of ASF by porting different security models from related work to ASF modules and by establishing a baseline for the impact of our infrastructure on the system performance.

The source code of our ANDROID SECURITY FRAMEWORK and our example modules can be anonymously retrieved from http://infsec.cs.uni-saarland.de/projects/asf/ASF_code.zip.

References

- [1] ABRAMS, M. D., EGGERS, K. W., LAPADULA, L. J., AND OLSON, I. M. A generalized framework for access control: An informal description. In *Proc. 13th National Computer Security Conference (NIST/NCSC)* (1990).
- [2] BACKES, M., GERLING, S., HAMMER, C., MAFFEI, M., AND VON STYP-REKOWSKY, P. Appguard - enforcing user requirements on Android apps. In *Proc. 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '13)* (2013).
- [3] BADGER, L., STERNE, D. F., SHERMAN, D. L., WALKER, K. M., AND HAGHIGHAT, S. A. A domain and type enforcement UNIX prototype. In *Proc. 5th conference on USENIX UNIX Security Symposium (SSYM '95)* (1995), USENIX Association.
- [4] BADGER, L., STERNE, D. F., SHERMAN, D. L., WALKER, K. M., AND HAGHIGHAT, S. A. Practical domain and type enforcement for UNIX. In *Proc. IEEE Symposium on Security and Privacy (SP '95)* (1995), IEEE Computer Society.
- [5] BAKER, D. B. Fortresses built upon sand. In *Proc. New security paradigms workshop (NSPW '96)* (1996), ACM.
- [6] BUGIEL, S., DAVI, L., DMITRIENKO, A., FISCHER, T., SADEGHI, A.-R., AND SHASTRY, B. Towards taming privilege-escalation attacks on Android. In *Proc. 19th Annual Network and Distributed System Security Symposium (NDSS '12)* (2012), The Internet Society.
- [7] BUGIEL, S., DAVI, L., DMITRIENKO, A., HEUSER, S., SADEGHI, A.-R., AND SHASTRY, B. Practical and lightweight domain isolation on Android. In *Proc. 1st ACM Workshop on Security and Privacy in Mobile Devices (SPSM '11)* (2011), ACM.
- [8] BUGIEL, S., HEUSER, S., AND SADEGHI, A.-R. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *Proc. 22nd USENIX Security Symposium (SEC '13)* (2013), USENIX Association.
- [9] CHIN, E., PORTER FELT, A., GREENWOOD, K., AND WAGNER, D. Analyzing inter-application communication in Android. In *Proc. 9th International Conference on Mobile Systems, Applications, and Services (MobiSys '11)* (2011), ACM.
- [10] CONTI, M., NGUYEN, V. T. N., AND CRISPO, B. CRePE: Context-related policy enforcement for android. In *Proc. 13th International Conference on Information Security (ISC '10)* (2010), Springer-Verlag.
- [11] DEPARTMENT OF DEFENSE. *Trusted Computer System Evaluation Criteria*. Dec. 1985. (Orange Book).
- [12] EDGE, J. The return of loadable security modules? Online: <http://lwn.net/Articles/526983/>, Nov. 2012.
- [13] EDWARDS, A., JAEGER, T., AND ZHANG, X. Runtime verification of authorization hook placement for the Linux security modules framework. In *Proc. 9th ACM Conference on Computer and Communication Security (CCS '02)* (2002), ACM.
- [14] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)* (2010), USENIX Association.
- [15] ENCK, W., ONGTANG, M., AND MCDANIEL, P. On lightweight mobile phone application certification. In *Proc. 16th ACM Conference on Computer and Communication Security (CCS '09)* (2009), ACM.
- [16] ERLINGSSON, Ú., AND SCHNEIDER, F. B. IRM enforcement of Java stack inspection. In *Proc. 23rd IEEE Symposium on Security and Privacy (SP '02)* (2000), IEEE Computer Society.
- [17] FRASER, T. LOMAC: MAC you can live with. In *Proc. FREENIX Track: 2001 USENIX Annual Technical Conference* (2001), USENIX Association.
- [18] FRASER, T., BADGER, L., AND FELDMAN, M. Hardening COTS software with generic software wrappers. In *Proc. IEEE Symposium on Security and Privacy (SP '99)* (1999).
- [19] GANAPATHY, V., JAEGER, T., AND JHA, S. Automatic placement of authorization hooks in the Linux Security Modules framework. In *Proc. 12th ACM Conference on Computer and Communication Security (CCS '05)* (2005), ACM.
- [20] GLIGOR, V., GAVRILA, S., AND FERRAILOLO, D. On the formal definition of separation-of-duty policies and their composition. In *Proc. 19th IEEE Symposium on Security and Privacy (SP '98)* (1998), IEEE Computer Society.
- [21] GRACE, M., ZHOU, W., JIANG, X., AND SADEGHI, A.-R. Unsafe exposure analysis of mobile in-app advertisements. In *Proc. 5th ACM conference on Security and Privacy in Wireless and Mobile Networks (WISEC '12)* (2012), ACM.
- [22] GUY, M. Jisc good apis management report: A review of good practice in the provision of machine interfaces and use of services, May 2009.
- [23] HARADA, T., HORIE, T., AND TANAKA, K. Task Oriented Management Obviates Your Onus on Linux. In *Linux Conference* (2004).
- [24] HORNACK, P., HAN, S., JUNG, J., SCHECHTER, S., AND WETHERALL, D. These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications. In *Proc. 18th ACM Conference on Computer and Communication Security (CCS '11)* (2011), ACM.
- [25] JEON, J., MICINSKI, K. K., VAUGHAN, J. A., FOGEL, A., REDDY, N., FOSTER, J. S., AND MILLSTEIN, T. Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android. In *Proc. 2nd ACM workshop on Security and privacy in smartphones and mobile devices (SPSM '12)* (2012), ACM.
- [26] LAMPSON, B. W. Protection. *ACM SIGOPS Operating Systems Review* 8, 1 (Jan. 1974), 18–24.
- [27] LIGATTI, J., BAUER, L., AND WALKER, D. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security* 4, 1–2 (2005), 2–16.
- [28] LINDEN, T. A. Operating system structures to support security and reliable software. *ACM Computer Surveys* 8, 4 (Dec. 1976), 409–445.
- [29] LINUX CROSS REFERENCE. Linux Security Module framework. Online: <http://lxr.free-electrons.com/source/Documentation/security/LSM.txt>.
- [30] LOSCOCO, P., AND SMALLEY, S. Integrating flexible support for security policies into the Linux operating system. In *Proc. FREENIX Track: 2001 USENIX Annual Technical Conference* (2001), USENIX Association.

- [31] LOSCOCO, P. A., SMALLEY, S. D., MUCKELBAUER, P. A., TAYLOR, R. C., TURNER, S. J., AND FARRELL, J. F. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proc. 21st National Information Systems Security Conference (NISSC '98)* (1998).
- [32] MCDANIEL, P., AND PRAKASH, A. Methods and limitations of security policy reconciliation. In *Proc. 23rd IEEE Symposium on Security and Privacy (SP '02)* (2002), IEEE Computer Society.
- [33] NADKARNI, A., AND ENCK, W. Preventing accidental data disclosure in modern operating systems. In *Proc. 20th ACM Conference on Computer and Communication Security (CCS '13)* (2013).
- [34] NAUMAN, M., KHAN, S., AND ZHANG, X. Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In *Proc. 5th ACM Symposium on Information, Computer and Communication Security (ASIACCS '10)* (2010), ACM.
- [35] ONGTANG, M., MCLAUGHLIN, S. E., ENCK, W., AND MCDANIEL, P. Semantically rich application-centric security in Android. In *Proc. 25th Annual Computer Security Applications Conference (ACSAC '09)* (2009), ACM.
- [36] OTT, A. The Rule Set Based Access Control (RSBAC) Linux Kernel Security Extension. In *8th International Linux Kongress* (2001).
- [37] PEARCE, P., PORTER FELT, A., NUNEZ, G., AND WAGNER, D. AdDroid: Privilege separation for applications and advertisers in Android. In *Proc. 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS '12)* (2012), ACM.
- [38] PORTER FELT, A., FINIFTER, M., CHIN, E., HANNA, S., AND WAGNER, D. A survey of mobile malware in the wild. In *Proc. 1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM '11)* (2011), ACM.
- [39] PORTER FELT, A., HA, E., EGELMAN, S., HANEY, A., CHIN, E., AND WAGNER, D. Android permissions: user attention, comprehension, and behavior. In *Proc. 8th Symposium on Usable Privacy and Security (SOUPS '12)* (2012), ACM.
- [40] PORTER FELT, A., WANG, H. J., MOSCHUK, A., HANNA, S., AND CHIN, E. Permission re-delegation: Attacks and defenses. In *Proc. 20th USENIX Security Symposium (SEC '11)* (2011), USENIX Association.
- [41] PROVOS, N. Improving host security with system call policies. In *Proc. 12th USENIX Security Symposium (SEC '03)* (2003), USENIX Association.
- [42] PROVOS, N., FRIEDL, M., AND HONEYMAN, P. Preventing privilege escalation. In *Proc. 12th USENIX Security Symposium (SEC '03)* (2003), USENIX Association.
- [43] RAO, V., AND JAEGER, T. Dynamic mandatory access control for multiple stakeholders. In *Proc. 14th Symposium on Access Control Models and Technologies (SACMAT '09)* (2009), ACM.
- [44] RUSSELLO, G., CONTI, M., CRISPO, B., AND FERNANDES, E. MOSES: supporting operation modes on smartphones. In *Proc. 17th Symposium on Access Control Models and Technologies (SACMAT '12)* (2012), ACM.
- [45] SALTZER, J., AND SCHROEDER, M. The protection of information in computer systems. *Proceedings of the IEEE* 63, 9 (1975), 1278–1308.
- [46] SHEKHAR, S., DIETZ, M., AND WALLACH, D. S. Ad-split: Separating smartphone advertising from applications. In *Proc. 21st USENIX Security Symposium (SEC '12)* (2012), USENIX Association.
- [47] SMALLEY, S., AND CRAIG, R. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proc. 20th Annual Network and Distributed System Security Symposium (NDSS '13)* (2013), The Internet Society.
- [48] SMALLEY, S., VANCE, C., AND SALAMON, W. Implementing SELinux as a Linux security module. NAI Labs Report #01-043, NAI Labs, Dec 2001. Revised May 2002.
- [49] SPENCER, R., SMALLEY, S., LOSCOCO, P., HIBLER, M., ANDERSEN, D., AND LEPREAU, J. The Flask security architecture: System support for diverse security policies. In *Proc. 8th USENIX Security Symposium (SEC '99)* (1999), USENIX Association.
- [50] WANG, Y., HARIHARAN, S., ZHAO, C., LIU, J., AND DU, W. Compac: Enforce component-level access control in Android. In *Proc. 4th ACM conference on Data and application security and privacy (CODASPY '14)* (2014), ACM.
- [51] WATSON, R., MORRISON, W., VANCE, C., AND FELDMAN, B. The TrustedBSD MAC Framework: Extensible kernel access control for FreeBSD 5.0. In *Proc. FREEINX Track: 2003 USENIX Annual Technical Conference* (2003), USENIX Association.
- [52] WATSON, R. N. M. New approaches to operating system security extensibility. Tech. Rep. UCAM-CL-TR-818, University of Cambridge, Computer Laboratory, Apr. 2012.
- [53] WRIGHT, C., COWAN, C., SMALLEY, S., MORRIS, J., AND KROAH-HARTMAN, G. Linux Security Modules: General security support for the Linux kernel. In *Proc. 11th USENIX Security Symposium (SEC '02)* (2002), USENIX Association.
- [54] XU, R., SAÏDI, H., AND ANDERSON, R. Aurasium: Practical policy enforcement for Android applications. In *Proc. 21st USENIX Security Symposium (SEC '12)* (2012), USENIX Association.
- [55] ZHOU, Y., AND JIANG, X. Dissecting Android malware: Characterization and evolution. In *Proc. 33rd IEEE Symposium on Security and Privacy (SP '12)* (2012), IEEE Computer Society.
- [56] ZHOU, Y., AND JIANG, X. Detecting passive content leaks and pollution in Android applications. In *Proc. 20th Annual Network and Distributed System Security Symposium (NDSS '13)* (2013), The Internet Society.
- [57] ZHOU, Y., ZHANG, X., JIANG, X., AND FREEH, V. Taming information-stealing smartphone applications (on Android). In *Proc. 4th International Conference on Trust and Trustworthy Computing (TRUST '11)* (2011), Springer-Verlag.

Appendix

A Policy Module Interface

Listing 2: *Interface for Access Control Policy Modules*

```
1 public interface IAccessControlModule {
2     /* *****
3      * General functions
4      * ***** */
5     public boolean init();
6     public ModuleConfiguration getConfig();
7     public void shutdown();
8
9     /* *****
10     * Package life-cycle event hooks
11     * ***** */
12     public void security_event_installNewPackage(PackageParser.Package pkg, UserHandle user);
13     public void security_event_replacePackage(PackageParser.Package oldPkg, PackageParser.Package newPkg, UserHandle
14         user);
15     public void security_event_deletePackage(String packageName, int uid, int removedAppId, int removedUsers[],
16         UserHandle user);
17
18     /* *****
19     * Generic hooks
20     * ***** */
21     public void security_generic_checkPolicy(Bundle arguments);
22     public void security_generic_callModule(Bundle arguments);
23     public boolean security_generic_instrumentApp(String packageName);
24
25     /* *****
26     * Broadcast hooks
27     * ***** */
28     public boolean security_broadcast_deliverToRegisteredReceiver(Intent intent, ComponentName targetComp, String
29         requiredPermission, int targetUid, int targetPid, String callerPackage, ApplicationInfo callerApp, int callingUid, int
30         callingPid);
31     public boolean security_broadcast_processNextBroadcast(Intent intent, ResolveInfo target, String requiredPermission,
32         String callerPackage, ApplicationInfo callerApp, int callingUid, int callingPid);
33
34     /* *****
35     * ContentProvider.Transport hooks
36     * ***** */
37     public boolean security_cp_applyOperation(ContentProviderOperation op, int uid, int pid);
38     public boolean security_cp_preQuery(String callingPkg, Uri uri, String[] projection, String selection, String[]
39         selectionArgs, String sortOrder, int uid, int pid);
40     public Cursor security_cp_postQuery(Cursor result, String callingPkg, Uri uri, String[] projection, String selection,
41         String[] selectionArgs, String sortOrder, int uid, int pid);
42     public boolean security_cp_insert(Uri uri, ContentValues initialValues, int uid, int pid);
43     public boolean security_cp_bulkInsert(Uri uri, ContentValues[] initialValues, int uid, int pid);
44     public boolean security_cp_delete(String callingPkg, Uri uri, String selection, String[] selectionArgs, int uid, int pid);
45     public boolean security_cp_update(String callingPkg, Uri uri, ContentValues values, String selection, String[]
46         selectionArgs, int uid, int pid);
47     public boolean security_cp_openFile(Uri uri, String mode, int uid, int pid);
48     public boolean security_cp_preCall(String providerClass, String method, String arg, Bundle extras, int uid, int pid);
49     public Bundle security_cp_postCall(Bundle result, String providerClass, String method, String arg, Bundle extras, int
50         uid, int pid);
51
52     public boolean security_contacts_preQueryDirectory(Uri uri, String directoryName, String directoryType, String[]
53         projection, String selection, String[] selectionArgs, String sortOrder, int uid, int pid);
54     public BulkCursorDescriptor security_contacts_postQueryDirectory(BulkCursorDescriptor result, String directoryName,
55         String directoryType, String providerName, Uri uri, String[] projection, String selection, String[] selectionArgs, String
56         sortOrder, int uid, int pid);
57
58     /* *****
59     * Activity related hooks
60     * ***** */
61 }
```

```

49 public boolean security_ams_startActivity(Intent intent, String resolvedType, ActivityInfo aInfo, String resultWho, int
    requestCode, int startFlags, Bundle options, ApplicationInfo callerInfo, int callingPid, int callingUid, int
    callingUserId);
50 public boolean security_ams_finishActivity(ComponentName origActivity, ComponentName realActivity, Intent intent,
    int userId, ApplicationInfo info, int resultCode, Intent resultData, int uid, int pid);
51 public boolean security_ams_moveTaskToFront(ComponentName origActivity, ComponentName realActivity, Intent
    intent, int userId, ApplicationInfo info, int flags, Bundle options, int uid, int pid);
52 public boolean security_ams_moveTaskToBack(ComponentName origActivity, ComponentName realActivity, Intent
    intent, int userId, ApplicationInfo info, int uid, int pid);
53 public boolean security_ams_clearApplicationUserData(String packageName, int pkgUid, int userId, int uid, int pid);
54
55 /* *****
56  * Permission check overrides
57  ***** */
58 public int security_ams_checkComponentPermission(String permission, int origUid, int origPid, int tlsUid, int tlsPid,
    int owningUid, boolean exported, int callerUid, int callerPid);
59 public boolean security_ams_checkCpuPermission(Uri uri, ProviderInfo cpi, int processUid, int processPid, boolean
    processIsolated, int processUserId, String processName, ApplicationInfo info, int uid, int pid);
60 public boolean security_ams_checkCpuPermission(Uri uri, ProviderInfo cpi, int uid, int pid);
61 public boolean security_ams_checkGrantUriPermission(int callingUid, String targetPkg, int targetUid, Uri uri, int
    modeFlags);
62 public int security_ams_checkUriPermission(Uri uri, int origUid, int origPid, int tlsUid, int tlsPid, int modeFlags);
63
64 /* *****
65  * PackageManagerService hooks
66  ***** */
67 public boolean security_pms_getPackageInfo(PackageInfo pi, int flags, int userId, boolean isUninstalled, int uid, int
    pid);
68 public boolean security_pms_getPackageUid(ApplicationInfo info, int userId, int uid, int pid);
69 public boolean security_pms_getPackageGids(ApplicationInfo info, int[] gids, int uid, int pid);
70 public String[] security_pms_getPackagesForUid(int forUid, String[] packages, int uid, int pid);
71 public boolean security_pms_getNameForUid(int forUid, String name, int uid, int pid);
72 public boolean security_pms_getUidForSharedUser(String sharedUserName, int suid, int uid, int pid);
73 public boolean security_pms_findPreferredActivity(Intent intent, String resolvedType, int flags, ResolveInfo ri, int
    priority, int userId, int uid, int pid);
74 public List<ResolveInfo> security_pms_queryIntentActivities(List<ResolveInfo> currentList, Intent intent, String
    resolvedType, int flags, int userId, int uid, int pid);
75 public List<ResolveInfo> security_pms_queryIntentReceivers(List<ResolveInfo> currentList, Intent intent, String
    resolvedType, int flags, int userId, int uid, int pid);
76 public List<ResolveInfo> security_pms_queryIntentServices(List<ResolveInfo> currentList, Intent intent, String
    resolvedType, int flags, int userId, int uid, int pid);
77 public ArrayList<PackageInfo> security_pms_getInstalledPackages(ArrayList<PackageInfo> currentList, int flags, int
    userId, int uid, int pid);
78 public ArrayList<PackageInfo> security_pms_getPackagesHoldingPermissions(ArrayList<PackageInfo> currentList, int
    flags, int userId, String[] permissions, int uid, int pid);
79 public ArrayList<ApplicationInfo> security_pms_getInstalledApplications(ArrayList<ApplicationInfo> currentList, int
    flags, int userId, int uid, int pid);
80 public ArrayList<ApplicationInfo> security_pms_getPersistentApplications(ArrayList<ApplicationInfo> currentList,
    int flags, int uid, int pid);
81 public boolean security_pms_getProviderInfo(ProviderInfo pi, ComponentName component, int flags, int userId, int
    uid, int pid);
82 public boolean security_pms_getActivityInfo(ActivityInfo ai, ComponentName component, int flags, int userId, int
    uid, int pid);
83 public boolean security_pms_getReceiverInfo(ActivityInfo ai, ComponentName component, int flags, int userId, int
    uid, int pid);
84 public boolean security_pms_getServiceInfo(ServiceInfo si, ComponentName component, int flags, int userId, int uid,
    int pid);
85 /* Pre-init function (packages are scanned before init is called) */
86 public boolean security_pms_scanPackage(PackageParser.Package pkg);
87 public boolean security_pms_deletePackage(PackageParser.Package pkg, boolean isSystemApp, boolean dataOnly,
    int flags);
88 public boolean security_pms_deletePackageSingleUser(PackageParser.Package pkg, boolean isSystemApp, int flags,
    int user);
89
90 /* *****
91  * Content Provider (general) related hooks
92  ***** */
93 /* Changed ProcessRecord to public for our module SDK */

```

```

94 public boolean security_ams_checkContentProviderPermission(ProviderInfo cpi, String permission, int processUid, int
    processPid, boolean procesIsolated, int processUserId, String processName, ApplicationInfo info, int uid, int pid);
95 public boolean security_ams_checkContentProviderPermission(ProviderInfo cpi, String permission, int uid, int pid);
96 public boolean security_ams_checkPathPermission(ProviderInfo cpi, PathPermission pp, String permission, int
    processUid, int processPid, boolean procesIsolated, int processUserId, String processName, ApplicationInfo info,
    int uid, int pid);
97 public boolean security_ams_checkPathPermission(ProviderInfo cpi, PathPermission pp, String permission, int uid, int
    pid);
98 public boolean security_ams_checkAppSwitchAllowed(int uid, int pid);
99
100 /* *****
101  * Service related hooks
102  ***** */
103 public List<ActivityManager.RunningServiceInfo>
    security_ams_getServices(ArrayList<ActivityManager.RunningServiceInfo> srvList, int uid, int pid);
104 public boolean security_ams_peekService(Intent service, String resolvedType, ServiceInfo serviceInfo, ApplicationInfo
    appInfo, String packageName, String permission, int uid, int pid);
105 public boolean security_ams_startService(Intent service, String resolvedType, ComponentName name, String
    shortName, ServiceInfo serviceInfo, ApplicationInfo appInfo, int srvUserId, String packageName, String
    processName, String permission, int callingPid, int callingUid);
106 public boolean security_ams_stopService(Intent service, String resolvedType, ComponentName name, String
    shortName, ServiceInfo serviceInfo, ApplicationInfo appInfo, int srvUserId, String packageName, String
    processName, String permission, int callingPid, int callingUid);
107 public boolean security_ams_bindService(Intent service, String resolvedType, int flags, ComponentName name, String
    shortName, ServiceInfo serviceInfo, ApplicationInfo appInfo, int srvUserId, String packageName, String
    processName, String permission, int callingPid, int callingUid);
108
109 /* *****
110  * LocationManagerService hooks
111  ***** */
112 public void security_location_getAllProviders(List<String> providerList, int uid, int pid);
113 public void security_location_getProviders(List<String> providers, Criteria criteria, boolean enabledOnly, int uid, int
    pid);
114 /* Unhide LocationRequest for our module SDK */
115 public void security_location_requestLocationUpdates(LocationRequest request, PendingIntent pi, int uid, int pid);
116 public void security_location_removeLocationUpdates(PendingIntent pi, int uid, int pid);
117 public Location security_location_getLastLocation(Location currentLocation, LocationRequest request, int uid, int pid);
118 public boolean security_location_addGpsStatusListener(int uid, int pid);
119 public boolean security_location_sendExtraCommand(String provider, String command, Bundle extras, int uid, int
    pid);
120 /* Unhide Geofence class for our module SDK */
121 public void security_location_requestGeofence(LocationRequest request, Geofence geofence, PendingIntent intent, int
    uid, int pid);
122 public void security_location_removeGeofence(Geofence geofence, PendingIntent intent, int uid, int pid);
123 public boolean security_location_isProviderEnabled(String provider, int uid, int pid);
124 public Location security_location_reportLocation(Location location, boolean passive, int uid, int pid);
125 public ProviderProperties security_location_addTestProvider(String name, ProviderProperties properties, int uid, int
    pid);
126 public boolean security_location_removeTestProvider(String provider, int uid, int pid);
127 public boolean security_location_setTestProviderLocation(String provider, Location location, int uid, int pid);
128 public boolean security_location_clearTestProviderLocation(String provider, int uid, int pid);
129 public boolean security_location_setTestProviderEnabled(String provider, boolean enabled, int uid, int pid);
130 public boolean security_location_clearTestProviderEnabled(String provider, int uid, int pid);
131 public boolean security_location_setTestProviderStatus(String provider, int status, Bundle extras, long updateTime,
    int uid, int pid);
132 public boolean security_location_clearTestProviderStatus(String provider, int uid, int pid);
133 public boolean security_location_sendLocationUpdate(Location location, String receiverPackageName, int pid, int uid);
134 public boolean security_location_updateFence(Location location, Geofence fence, PendingIntent fenceIntent, String
    fencePackageName, int uid);
135
136 /* *****
137  * AudioService hooks
138  ***** */
139 public boolean security_audio_adjustStreamVolume(int streamType, int direction, int flags, int uid, int pid);
140 public boolean security_audio_setStreamVolume(int streamType, int index, int flags, int uid, int pid);
141 public boolean security_audio_setMasterVolume(int volume, int flags, int uid, int pid);
142 public boolean security_audio_setRingerMode(int mode, int uid, int pid);
143 public boolean security_audio_setSpeakerphoneOn(boolean on, int uid, int pid);

```

```

144
145 /* *****
146 * TelephonyService hooks
147 ***** */
148 public boolean security__telephony__call(String number, int uid, int pid);
149 public List<NeighboringCellInfo> security__telephony__getNeighboringCellInfo(List<NeighboringCellInfo> currentList,
    int uid, int pid);
150
151 /* *****
152 * SMS and MMS Service hooks
153 ***** */
154 public boolean security__sms__copyMessageToIcc(int status, byte[] pdu, byte[] smsc, int uid, int pid);
155 public boolean security__sms__getAllMessagesFromIcc(int uid, int pid);
156 public List<RawByteData> security__sms__getAllMessagesFromIccFilter(List<RawByteData> rawSms, int uid, int pid);
157 public boolean security__sms__sendData(String destAddr, String scAddr, int destPort, byte[] data, PendingIntent
    sentIntent, PendingIntent deliveryIntent, int uid, int pid);
158 public boolean security__sms__sendText(String destAddr, String scAddr, String text, PendingIntent sentIntent,
    PendingIntent deliveryIntent, int uid, int pid);
159 public boolean security__sms__sendMultipartText(String destAddr, String scAddr, List<String> parts,
    List<PendingIntent> sentIntents, List<PendingIntent> deliveryIntents, int uid, int pid);
160 public boolean security__sms__updateMessageOnIccEf(int index, int status, byte[] pdu, int uid, int pid);
161
162 /* *****
163 * WiFi Service hooks
164 ***** */
165 public List<ScanResult> security__wifi__getScanResult(List<ScanResult> result, int uid, int pid);
166 public boolean security__wifi__startScan(int uid, int pid);
167 public boolean security__wifi__setWifiEnabled(boolean enable, int uid, int pid);
168 public boolean security__wifi__setWifiApEnabled(WifiConfiguration wifiConfig, boolean enabled, int uid, int pid);
169 public boolean security__wifi__setWifiApConfiguration(WifiConfiguration wifiConfig, int uid, int pid);
170 public boolean security__wifi__disconnect(int uid, int pid);
171 public boolean security__wifi__reconnect(int uid, int pid);
172 public boolean security__wifi__reassociate(int uid, int pid);
173 public List<WifiConfiguration> security__wifi__getConfiguredNetworks(List<WifiConfiguration> currentList, int uid, int
    pid);
174 public boolean security__wifi__addOrUpdateNetwork(WifiConfiguration config, int uid, int pid);
175 public boolean security__wifi__removeNetwork(int netId, int uid, int pid);
176 public boolean security__wifi__enableNetwork(int netId, boolean disableOthers, int uid, int pid);
177 public boolean security__wifi__disableNetwork(int netId, int uid, int pid);
178 public boolean security__wifi__getConnectionInfo(WifiInfo info, int uid, int pid);
179 public boolean security__wifi__setCountryCode(String countryCode, boolean persist, int uid, int pid);
180 public boolean security__wifi__setFrequencyBand(int band, boolean persist, int uid, int pid);
181 public boolean security__wifi__startWifi(int uid, int pid);
182 public boolean security__wifi__stopWifi(int uid, int pid);
183 public boolean security__wifi__addToBlacklist(String bssid, int uid, int pid);
184 public boolean security__wifi__clearBlacklist(int uid, int pid);
185 public boolean security__wifi__getWifiServiceMessenger(int uid, int pid);
186 public boolean security__wifi__getWifiStateMachineMessenger(int uid, int pid);
187 public boolean security__wifi__getConfigFile(String currentConfig, int uid, int pid);
188
189 /* *****
190 * ClipboardService hooks
191 ***** */
192 public ClipData security__clip__getPrimaryClip(ClipData currentPrimary, int clipUid, int uid, int pid);
193 public boolean security__clip__setPrimaryClip(ClipData clip, int uid, int pid);
194 public boolean security__clip__informPrimaryClipChanged(ClipData currentPrimary, int setByUid, String packageName,
    int uid);
195 public ClipDescription security__clip__getPrimaryClipDescription(ClipDescription currentDescription, int clipUid, int uid,
    int pid);
196 public boolean security__clip__hasPrimaryClip(boolean hasClipboard, int clipUid, int uid, int pid);
197 public boolean security__clip__hasClipboardText(String currentText, int clipUid, int uid, int pid);
198
199 /* *****
200 * PowerManagerService hooks
201 ***** */
202 public boolean security__power__acquireWakeLock(String tag, WorkSource ws, int uid, int pid);
203 public boolean security__power__userActivity(long eventTime, int event, int flags, int uid, int pid);
204 public boolean security__power__goToSleep(long eventTime, int reason, int uid, int pid);

```

```

205 public boolean security_power_wakeUp(long eventTime, int uid, int pid);
206 public boolean security_power_nap(long time, int uid, int pid);
207 public boolean security_power_setBacklightBrightness(int brightness, int uid, int pid);
208 public boolean security_power_reboot(boolean confirm, String reason, boolean wait, int uid, int pid);
209
210 /* *****
211  * PhoneSubscriberInfo hooks
212  * ***** */
213 public String security_phonesubinfo_getDeviceId(String id, int uid, int pid);
214 public String security_phonesubinfo_getDeviceSvn(String svn, int uid, int pid);
215 public String security_phonesubinfo_getSubscriberId(String id, int uid, int pid);
216 public String security_phonesubinfo_getGroupIdLevel1(String groupid, int uid, int pid);
217 public String security_phonesubinfo_getIccSerialNumber(String icc, int uid, int pid);
218 public String security_phonesubinfo_getLine1Number(String number, int uid, int pid);
219 public String security_phonesubinfo_getLine1AlphaTag(String tag, int uid, int pid);
220 public String security_phonesubinfo_getMsisdn(String msisd, int uid, int pid);
221 public String security_phonesubinfo_getVoiceMailNumber(String number, int uid, int pid);
222 public String security_phonesubinfo_getVoiceMailAlphaTag(String tag, int uid, int pid);
223 public String security_phonesubinfo_getIsimImpi(String impi, int uid, int pid);
224 public String security_phonesubinfo_getIsimDomain(String domain, int uid, int pid);
225 public String[] security_phonesubinfo_getIsimImpu(String impu[], int uid, int pid);
226 }

```

Listing 3: Interface for Access Control Policy Modules to Linux Security Module

```

1 public interface KMACAdaptor {
2     public boolean init();
3     public boolean isEnabled();
4     public boolean isEnforcing();
5     public boolean setEnforcing(boolean value);
6     public boolean setContext(String path, Bundle context);
7     public boolean restoreContext(Bundle context);
8     public Bundle getContext(String path);
9     public Bundle getPeerContext(FileDescriptor fd); /* wrapper around getsockopt call to LSM */
10    public Bundle getCurrentContext();
11    public Bundle getProcessContext(int pid);
12    public Bundle getConfig(Bundle args); /* e.g., get list of defined booleans or one specific boolean value */
13    public boolean setConfig(Bundle conf); /* e.g., set a boolean value */
14    public boolean checkAccess(Bundle args); /* args can be, e.g., quadruple of subject ctx, object ctx, object class, op */
15
16    /* Zygote is statically integrated with the Kernel MAC, thus, each KMACAdaptor must implemented these hooks in
17       ZygoteConnection */
18    public boolean security_zygote_applyUidSecurityPolicy(Credentials creds, Bundle peerSecurityContext);
19    public boolean security_zygote_applyRlimitSecurityPolicy(Credentials creds, Bundle peerSecurityContext);
20    public boolean security_zygote_applyCapabilitiesSecurityPolicy(Credentials creds, Bundle peerSecurityContext);
21    public boolean security_zygote_applyInvokeWithSecurityPolicy(Credentials creds, Bundle peerSecurityContext);
22    public boolean security_zygote_applySecurityLabelPolicy(Credentials creds, Bundle peerSecurityContext);
23 }

```

Listing 4: Methods for IRM instrumentation

```

1 public class Instrumentation {
2     public static void initClass(Class<?> clazz);
3
4     public static int redirectMethod(String fromDescriptor, String toDescriptor);
5     public static int redirectMethod(Signature from, Signature to);
6
7     public static void callVoidMethod(Class<?> caller, Object _this, Object... args);
8     public static void callVoidMethod(String id, Object _this, Object... args);
9     public static void callVoidMethod(int methodId, Object _this, Object... args);
10    public static int callIntMethod(Class<?> caller, Object _this, Object... args);
11    public static int callIntMethod(String id, Object _this, Object... args);
12    public static int callIntMethod(int methodId, Object _this, Object... args);
13    public static boolean callBooleanMethod(Class<?> caller, Object _this, Object... args);
14    public static boolean callBooleanMethod(String id, Object _this, Object... args);
15    public static boolean callBooleanMethod(int methodId, Object _this, Object... args);
16    public static Object callObjectMethod(Class<?> caller, Object _this, Object... args);
17    public static Object callObjectMethod(String id, Object _this, Object... args);
18    public static Object callObjectMethod(int methodId, Object _this, Object... args);

```

```

19 public static void callStaticVoidMethod(Class<?> caller, Class<?> _clazz, Object... args);
20 public static void callStaticVoidMethod(String id, Class<?> _clazz, Object... args);
21 public static void callStaticVoidMethod(int methodId, Class<?> _clazz, Object... args);
22 public static Object callStaticObjectMethod(Class<?> caller, Class<?> _clazz, Object... args);
23 public static Object callStaticObjectMethod(String id, Class<?> _clazz, Object... args);
24 public static Object callStaticObjectMethod(int methodId, Class<?> _clazz, Object... args);
25 }

```

B Break Down of Policy Enforcement Coverage

System App/Service	Number of hooks	Example hooks
BroadcastQueue	2	deliverToRegisteredReceiver, processNextBroadcast
ContentProvider	12	insert, update, preQuery, postQuery
ActivityStack	5	startActivity, moveTaskToBack, finishActivity
ActivityManagerService	10	checkComponentPermission, checkUriPermission, checkGrantUriPermission
PackageManagerService	21	getPackageInfo, findPreferredActivity, queryIntentReceivers, getServiceInfo, scanPackage, deletePackage
ActiveServices	5	startService, bindService, getServices
LocationManagerService	21	getProviders, requestLocationUpdates, requestGeofence, reportLocation, setTestProviderLocation
AudioService	5	adjustStreamVolume, setMasterVolume, setRingerMode
TelephonyService	2	call, getNeighboringCells
SMSService	7	getAllMessagesFromIcc, sendData, sendText
WiFiService	23	getScanResults, addOrUpdateNetwork, getConnectionInfo, getWifiServiceManager
ClipboardService	7	getPrimaryClip, setPrimaryClip
PowerManagerService	6	acquireWakeLock, userActivity, reboot
PhoneSubInfo	13	getDeviceId, getIccSerialNumber, getLine1Number, getIsimImpi
Total:	139	

Table 3: Break down of hooked system apps and services.

C Break Down of Most Frequently Called Enforcement Hooks

Hooked function	Android Security Framework		Stock Android v4.3	
	Frequency	Mean (μ s)	Frequency	Mean (μ s)
ActivityManagerService.checkComponentPermission	1705	39.413 \pm 0.658	2024	36.518 \pm 0.523
BroadcastQueue.broadcastIntent	908	305.274 \pm 16.752	1007	332.328 \pm 17.085
SettingsProvider.call	544	67.710 \pm 3.004	669	46.574 \pm 1.723
PackageManagerService.queryIntentReceivers	438	92.458 \pm 3.598	745	84.343 \pm 2.296
PackageManagerService.queryIntentActivities	296	192.178 \pm 15.458	242	195.211 \pm 18.355
PowerManagerService.acquireWakeLock	229	296.246 \pm 10.740	255	295.601 \pm 11.121
PackageManagerService.getActivityInfo	229	53.039 \pm 2.223	203	45.551 \pm 2.104
PackageManagerService.getPackageInfo	207	47.324 \pm 2.339	307	37.774 \pm 1.456
PackageManagerService.queryIntentServices	123	131.744 \pm 9.220	134	106.354 \pm 6.069
PackageManagerService.getPackageUid	93	35.767 \pm 2.353	201	30.005 \pm 0.000

Table 4: Ten most frequently invoked hooked functions and their average performance overhead on ANDROID SECURITY FRAMEWORK vs. stock Android v4.3. The margins of error are given for the 95% confidence interval.

D Module Performance

Table 5 provides an overview of the performance impact of different security models as reported in their respective publications (*native*) and as measured by us for their implementation as module (*ASF Module*). However, it should be noted, that these are not directly comparable, because all security models have originally been implemented for a different Android OS version and been tested on a different hardware platform. Figure 8 presents the cumulative frequency distribution for the measured performance overhead of our example modules versus stock Android v4.3.

Use-case	Implementation	Android version	Test device	Average (μs)
CRePE [10] [*]	Native	v2.3	HTC Magic	≈ 100
	ASF Module [‡]	v4.3	Nexus 7	168.943 \pm 5.884
XManDroid [6]	Native [◊]	v2.2	Nexus One	532
	ASF Module [‡]	v4.3	Nexus 7	206.062 \pm 5.573
FlaskDroid [8] (middleware) [†]	Native	v4.0.3	Galaxy Nexus	452
	ASF Module [‡]	v4.3	Nexus 7	359.317 \pm 11.015

Table 5: *Performance measurements of our example modules.*

^{*} Two rules loaded. [◊] Weighted average for cached and uncached checks. [†] With basic policy loaded.

[‡] Weighted average incl. IPC roundtrip between hook and module.

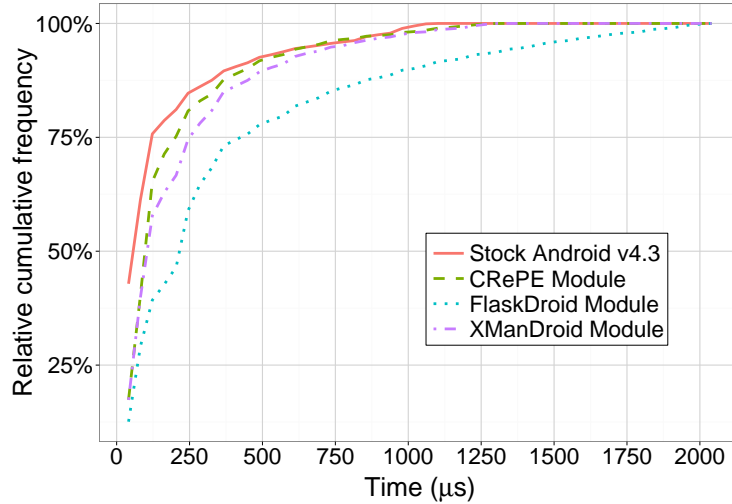


Figure 8: *Relative cumulative frequency distribution of example modules' performance overhead vs. stock Android v4.3.*

E Policy-agnostic Calls from Zygote to Linux Security Module

We briefly explain at the example of Zygote how the generic interface for calls to the kernel module can be used. As described in Section 5, we added a generic interface implementation, called `KMAC.java`, to the Android API. `KMAC.java` implements the interface described in Listing 3 in Appendix A. `KMAC.java` in turn loads via the Java reflection API the `LSM.java` classes and via JNI the `liblsm.so` deployed by modules and uses them to forward calls to the kernel module. `LSM.java` must hence also implement the interface in Listing 3. `LSM.java` and `liblsm.so` are responsible for translating the function arguments to the kernel module specific protocol. For instance, consider Listing 5 that shows how the KMAC interface is used in Zygote to verify that the caller is allowed to specify certain parameters like the UID/GID of a new app process. It first uses the `getPeerContext` function to retrieve the kernel-level security context of the calling process. This information is stored in a generic Bundle structure. It afterwards uses this information to request policy decisions from the Linux security module such as `security_zygote_applyUidSecurityPolicy` (a Zygote-specific hook). Listing 6 shows how the SE Android module (cf. Section 6) implements the interface to translate the arguments to SELinux-specific arguments and to call the SELinux kernel module. Here, `SELinux.java` takes the role of `LSM.java` and `SELinux.getPeerContext` and `SELinux.checkSELinuxAccess` are *native* functions that call via `libselinux.so` the kernel module. Hence, `libselinux.so` takes the role of `liblsm.so`.

Listing 5: *Use of generic Kernel module interface in ZygoteConnection.java*

```

1 private final Bundle peerSecurityContext;
2 private static final KMAC mKMAC = new KMAC();
3 ...
4 ZygoteConnection(LocalSocket socket) throws
   IOException {
5 ...
6     peerSecurityContext =
7         mKMAC.getPeerContext(mSocket.getFileDescriptor());
8 ...
9 ...
10 private static void applyUidSecurityPolicy(Arguments
    args, Credentials peer, Bundle
    peerSecurityContext) {
11 ...
12     boolean allowed =
13         mKMAC.security_zygote_applyUidSecurityPolicy(peer,
14             peerSecurityContext);

```

Listing 6: *Implementation of generic LSM interface for SELinux kernel module*

```

1 package android.os;
2
3 public class SELinuxAdaptor implements
   KMACAdaptor {
4 ...
5     @Override
6     public Bundle getPeerContext(FileDescriptor fd) {
7         String ctx = SELinux.getPeerContext(fd);
8         Bundle ret = new Bundle();
9         ret.putString("selinux.context", ctx);
10        return ret;
11    }
12
13    @Override
14    public boolean
        security_zygote_applyUidSecurityPolicy(Credentials
        creds, Bundle peerSecurityContext) {
15        String peerCtx =
16            peerSecurityContext.getString("selinux.context");
17        return SELinux.checkSELinuxAccess(peerCtx,
18            peerCtx, "zygote", "specifyids");
19    }

```

F Usage of Instrumentation API

We use the abstract example shown in Listing 7 to illustrate the usage of our instrumentation API. Calling `Instrumentation.redirectMethod()` (line 4) diverts the control from method `foo()` of class `com.test.A` to the method `bar()` of class `com.test.B`. It returns a reference to the original function, which we store in a variable `A_foo`. Subsequent calls to `A.foo()` (line 9) will invoke `B.bar()` instead. The original method `foo()` can still be invoked by calling `Instrumentation.callOriginalMethod(A_foo)` (line 11).

Listing 7: *Usage of the IRM instrumentation library*

```

1 public static void main(String[] args) {
2     A.foo(); // calls A.foo()
3
4     MethodHandle A_foo =
5         Instrumentation.redirectMethod(
6             "com.test.A->foo()",
7             "com.test.B->bar()");
8     // calls B.bar():
9     A.foo();
10    // calls A.foo():
11    Instrumentation.callOriginalMethod(A_foo);
12 }

```

G Details on FlaskDroid Hook Logic

We illustrate at the example of FlaskDroid [8] how the hook logic of existing solutions can be moved

into a module. Listing 8 shows one of the original FlaskDroid hooks in the `getAllProviders` function of the Android location service. The hook calls via the service’s context to FlaskDroid’s policy server where the access control decision is determined by the `checkPolicy` function. This function internally determines the subject’s and object’s security type from their UIDs⁶. A denial of access results always in a security exception that is thrown back to the caller of the location service API. Listing 9 shows the re-implementation of this logic in a module for our ANDROID SECURITY FRAMEWORK by simply overriding the corresponding enforcement function for `LocationManagerService.getAllProviders` and directly calling the `checkPolicy` function with all required parameters provided by the hook. It should be noted that in FlaskDroid the security exception in case of denied access is hardcoded in the system, while in an implementation as a module the FlaskDroid authors could alternatively change the enforcement to a less interruptive enforcement by re-assigning the `providerList` parameter to an empty list of Strings, i.e., pretending to the calling app that there is no location provider present in the system. In fact, as a module, such a change in strategy can be more easily rolled out than as a hardcoded implementation within the middleware.

Listing 8: *Original FlaskDroid hook in `com.android.server.LocationManagerService`*

```

1 public List<String> getAllProviders() {
2 ...
3 if(mContext.checkSecurityContext(Binder.getCallingUid(),
4   Process.myUid(), "locationService_c",
5   "getAllProviders") !=
6   PackageManager.PERMISSION_GRANTED) {
7   throw new SecurityException("Denied by MAC
8     policy");
9 }
10 ...

```

Listing 9: *Re-implementation of the hook from Listing 8 in a security module*

```

1 @Override
2 public void
3   security_location_getAllProviders(List<String>
4   providerList, int uid, int pid) {
5   if(checkPolicy(uid, Binder.getCallingUid(),
6     "locationService_c", "getAllProviders") ==
7     PackageManager.PERMISSION_DENIED) {
8     throw new SecurityException("Denied by MAC
9       policy");
10 }
11 }

```

⁶This is the original FlaskDroid behavior that is, as mentioned in Section 6.4, flawed.

H Further Security Modules

In this Section we briefly explain further example use-cases from related work that we ported as security modules in case their source code was available to us or that we re-implemented according to their published descriptions.

H.1 AppOps and IntentFirewall

Google introduced (unofficially) with Android v4.3 the *AppOps* infrastructure for dynamic, more fine-grained Permissions. It introduced hooks in different system services and apps, which query a central `AppOpsService` whether an application is allowed to perform an operation (e.g., retrieving the location of the device or querying a `ContentProvider`). The *AppOps* rules define a mapping from UID/package name to allowed operations. *AppOps* offers an interface to apps to retrieve the current configuration. Additionally, Google introduced (again unofficially) an *IntentFirewall*, which acts as a reference monitor for certain Intent-based operations like starting an `Activity`. The *IntentFirewall* rules describe which caller is allowed to receive which kind of Intent object, using the Intent’s attributes such as destination component. The sending or processing of Intents that violate these rules is aborted.

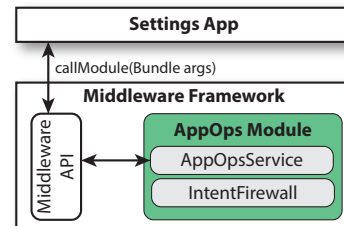


Figure 9: *AppOps and IntentFirewall module*

Implementation as a module: We ported *AppOps* and *IntentFirewall* (from Android v4.3) to a security module for ANDROID SECURITY FRAMEWORK (cf. Figure 9) by moving the `AppOpsService` and the `IntentFirewall` classes into a module. Our module comprises 2290 lines of code and differs in 33.71% of all LoC from the native implementation. The bulk of the changes (520 LoC), were required to move the hook logic of both services from the system apps and services of Android into the module by using our enforcement functions. For the *IntentFirewall*, this was straightforward and we only had to substitute a direct callback from *IntentFirewall* to the *ActivityManagerService* by our ASF callback mechanism. For

the AppOpsService, we had to add a mapping from caller PID to package name. By default the hooks of AppOps determine the caller’s package name and pass this information to the AppOpsService for policy check. Since this is a policy-specific logic of the hooks, our framework hooks do not (by default) provide the caller’s package name and we re-implemented this logic in our module by using our callback interface, which allows us to retrieve the package name for app PIDs. Moreover, we adapted the AppOpsService interface to retrieve/configure the current policies to a Bundle-based communication (e.g., we enabled the *PackageOps* and *OpEntry* to be serialized into a Bundle). AppOps is, furthermore, partially integrated into the *Settings* application to allow users to disable notifications from selected apps. We replaced this policy-specific channel between Settings and AppOps also with our policy-agnostic Bundle-based communication. Modules that support this Settings option, can return a value indicating whether notifications are disabled or not. If the module does not support this feature, Settings app by default allows notifications. However, our AppOps module does currently not support the operation watching feature, which requires the registration of application callback objects with the module.

H.2 Saint [35]

Saint is an extension for Android OS v1.5 that allows app developers to ship their apps with policy rules that determine how the app can interact with other apps in the system. For instance, the rules can declare that only apps with a specific package name, version, or set of permissions are allowed to call the app or be called by the app. The rules also support defining Intent attributes as rule criteria. The rules are enforced by the system through hooks in different system apps and services, such as the *ActivityManagerService*, that allow monitoring operations for the different app component types. Additionally, Saint provides a front-end app (*FrameworkPolicyManager*), that allows the user to override developer policies.

Implementation as a module: We re-implemented Saint as a security module by developing a module (729 LoC) that supports Saint’s policy language as described in [35]. We use our event functions to extract policy files from newly installed application packages and insert them into a policy database in our module. We use different hooks in the *ActivityManagerService* (e.g., starting an Activity, resolving an Activity, finding active services), Broadcast subsystem, or ContentProvider

class to enforce the Saint *runtime* policies. Using the *scanPackage* hook in the *PackageManagerService* we enforce Saint *install-time* policies to decide whether a new app is installed. Communication between the module the front-end app is again implemented based on Bundles. We successfully verified our Saint module’s effectiveness using the policies for the running example described by the Saint author’s [35] and a set of test apps that implement Saint’s example scenario.

H.3 TrustDroid [7]

TrustDroid extends the Android OS v2.2 architecture with isolation of different domains such as “work” and “private”. Every application is classified during installation into one of the available domains. For classification, TrustDroid uses package-specific attributes such as developer signature, external signature, or package name. Enforcement hooks at middleware and kernel level prevent at runtime any communication between different domains. At kernel level, TrustDroid uses TOMOYO Linux to enforce the policies. Policy rules for newly classified apps are propagated from the middleware to the kernel.

Implementation as a module: To re-implement TrustDroid as a security module (862 lines of code), we deployed a TOMOYO-enabled Linux kernel on the device (i.e., our KERNEL SUB-MODULE) and developed a MIDDLEWARE SUB-MODULE that deploys the required *LSM.java* and *libccs.so* to communicate with the kernel module. Additionally, we used our *scanPackage* hook in the *PackageManagerService* to classify newly installed applications and keep a mapping from UID to domain.⁷ Because TrustDroid’s policy is static and very simple, its architecture does not distinguish between policy enforcement and policy decision point, but instead every hook retrieves the domain of the current subject and object and denies access if their domains differ. We re-implemented this logic using the enforcement functions of our module, which was a straightforward implementation. For ContentProviders (e.g., Contacts) TrustDroid classifies the database entries and returns on access only the entries that have the same domain as the caller. Since we prohibit by design such as policy-specific intrusions into the default ContentProviders, we use our pre-query hooks to modify selection arguments to retrieve only contacts that are allowed for the current caller (e.g., where the contact’s group indicates a private contact). Using two example applications that

⁷TrustDroid does not allow apps in a shared sandbox to be classified differently.

are classified differently, we verified the effectiveness of our TrustDroid module.

H.4 Data shadowing [24, 57]

Both *AppFence* [24] as well as *TISSA* [57] provide a data shadowing feature. Data shadowing means, that an application that wants to retrieve sensitive information (e.g., contacts information, IMEI number, or location data) only get empty, fake, or filtered data.

Implementation as a module: We *re-implemented* the data shadowing features of AppFence and TISSA as a module by using our *edit automaton* hooks in the `ContentProvider.Transport` class, the `ContactsProvider`-specific hooks, Telephony service and Location service. For `ContentProvider` and `ContactsProvider`, in particular our pre-query and post-query hooks allowed us a fine-grained filtering or replacing (faking) of the returned data as well as returning an empty data set. However, the current coverage of our enforcement hooks does not include some of the data shadowing points of AppFence, such as microphone, logs, or camera, and we plan on adding them in the future.

H.5 Kirin [15]

Kirin extends Android’s application installation process with policy-based checks and denies installation of a new app when it violates the policy. Based on its time of publication, we presume that it was developed for Android OS v1.5.⁸ The actual policy check was performed in a dedicated Android application developed for Kirin, which interacted with the installation process. These policies are based on the set of permissions requested by an app and the interfaces (e.g., Broadcast receivers) it wants to register in the system. The installation of apps that are rejected by the policy is denied.

Implementation as a module: To *re-implement* Kirin’s security service as a security module, we developed a module that supports Kirin’s security language. Using our `scanPackage` hook in the *Package-ManagerService*, we check new applications against the policy and abort their installation in case the policy rejects the application. Our Kirin module comprises 246 lines of code.

⁸http://en.wikipedia.org/wiki/Android_version_history